Refinement Laws for Verifying Library Subroutine Adaptation

Colin Fidge Peter Robinson
School of Information Technology and Electrical Engineering
The University of Queensland, Australia
{cjf,pjr}@itee.uq.edu.au

Steve Dunne
School of Computing and Mathematics
The University of Teesside, United Kingdom
s.e.dunne@tees.ac.uk

Abstract

In Component-Based Software Engineering programs are constructed from pre-defined software library modules. However, if the library's subroutines do not exactly match the programmer's requirements, the subroutines' code must be adapted accordingly. For this process to be acceptable in safety or mission-critical applications, where all code must be proven correct, it must be possible to verify the correctness of the adaptations themselves. In this paper we show how refinement theory can be used to model typical adaptation steps and to define the conditions that must be proven to verify that a library subroutine has been adapted correctly.

1 Introduction

In Component-Based Software Engineering programs are constructed from predefined modules (or classes) from a software library, where each module encapsulates a number of variables (attributes) and subroutines (methods). Each subroutine has an advertised set of capabilities, and these must be matched against the programmer's specific computational requirements. Unfortunately, it is often the case that no existing library subroutine satisfies a given requirement exactly.

A commonly-suggested solution is to *adapt* a closely-matching subroutine by augmenting its body with additional program code that corrects the mismatch between the requirement and the subroutine's capabilities. In safety or mission-critical applications, however, where utmost guarantees of system correctness are needed, it must be possible to formally verify that all program code is correct, including any 'adaptation' (or 'gluing' [11] or 'wrapper' [19]) code. For instance, 'autocoding' tools are considered unsuitable for avionics software development unless they have a rigorous semantic theory for verifying the generated code's correctness [15].

Therefore, this paper presents a semantic foundation for verifiably-correct adaptation of software library subroutine bodies. This is done by modelling the programmer's requirement, the library subroutine code and the adaptation code semantically, and by then using program refinement theory to derive the conditions under which the adapted code meets the requirement. The resulting 'adaptation rules' formalise correct subroutine adaptation principles in a generic way.

2 Previous Work

The theory developed in this paper is motivated by previous research into constructing and using software module libraries.

In practice, large libraries of commonly-used data abstractions are now available for programming languages such as Ada [8]. In the past users of such libraries relied on informal keyword-based searches to find suitable modules [13], but libraries intended for high-integrity applications may also include formal pre and post-condition specifications of library subroutines [6]. Formal subroutine specifications, coupled with a program refinement theory, allow the correctness of a library module's implementation to be verified [4].

Theories for finding library components via formal specifications have been well explored in the literature. In particular, Zaremski and Wing's specification 'matching rules' [20] have been highly influential and form the starting point for our work on adaptation. Also, library modules are usually parameterised, so a simple form of adaptation occurs when the parameters are instantiated—most recently,

Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)
1530-0803/05 $20.00 © 2005 **IEEE**

theories have been developed for higher-order, subroutine-valued parameters [9].

More substantial forms of adaptation aim to change the behaviour of library components in ways not anticipated by their original designer. For instance, Inverardi and Tivoli show how the CCS process algebra can be used to derive 'gluing' code for composing library processes in order to satisfy desired properties [11]. Similarly, Spitznagel and Garlan use the FSP process algebra to model changes to interface protocols [19]. Roop *et al.* use Labelled Transition Systems as the formal basis for their algorithm for adapting embedded systems components [17]. Similarly, Bracciali *et al.* use the π-calculus to describe component adaptors [3]. However, all of this previous work focusses on concurrent *process* (or object) interfaces, whereas our research focusses on adapting imperative program code.

Closer to our research is Frappier *et al.*'s formalisation of a programming paradigm in which distinct requirements are satisfied separately and the resulting program fragments are then composed [5]. Like us, they use the refinement calculus as their formalism, but their development approach is top-down where ours is bottom-up. Haack *et al.* also consider 'adaptation' of formally-specified components, but their interest is the system maintenance problem caused by changes to library module *interfaces* [7]. Finally, research on software synthesis provides inspiration for our work, especially where it has a formal basis [12, 18], but this early work does not usually focus on subroutine adaptation *per se*.

Like us, Ayed *et al.* [1] have explored a refinement-based approach to program adaptation, but their formalism is more abstract than ours. They define fundamental semantic principles of, for example, *software modification* and *software composition*, rather than rules for producing specific programming language constructs. Also, they use a relational semantics where we use predicate transformers.

The approach most closely related to our work is Penix and Alexander's 'tactics' for embedding an inadequate subroutine in a software architecture that compensates for its shortcomings [16]. They describe two especially useful tactics: *choosing* between the inadequate subroutine and an alternative action, and *preceding* the subroutine with another action that completes the required behaviour. Our rules below incorporate both of these concepts, and add others.

3 Background: Program Refinement

In this section we briefly summarise notations from the refinement literature which are used to define and justify the new rules presented in Section 4. Refinement calculi come in a number of slightly different styles; herein we use Morgan's notation [14].

Program refinement calculi provide verifiably-correct laws for transforming specifications into executable programming language statements. The target language normally includes familiar programming language constructs for assignment (**:=**), sequential composition (**;**), choice (**if**), iteration (**while**), and variable declaration (**var**).

To express requirements to be implemented, the calculus also includes statements for declaration of logical constants (**con**) [14, §6.1], and specifications of the form '$v{:}[P\ ,\ Q]$' [14, §1.4.3]. Here v is a list of variable identifiers defining those variables that may be changed, P is a precondition predicate defining the expected program state when this statement is reached, and Q is a postcondition predicate defining the statement's intended effect on the state. Predicate Q can be used to relate the program's state before and after execution of the specification statement—a zero subscript 'v_0' is used to denote the initial value of a variable v, and the undecorated name 'v' is used for its final value [14, §6.2].

Program refinement is then embodied as a set of laws for translating specification statements into executable ones. This is denoted '$S_1 \sqsubseteq S_2$', meaning that (program) statement S_2 is a refinement of (specification) statement S_1. A refinement S_2 of some statement S_1 may be more deterministic than S_1 and may terminate more often. Numerous refinement laws are available [14]; Appendix A lists those particular laws needed in Section 4.

4 Formal Rules for Adapting Subroutines

In this section we define and verify correct our new library code adaptation rules using program refinement principles. In each case the goal is to replace a specified requirement appearing in a program under construction with executable code produced by adapting the body of a subroutine from a software library.

We assume that the requirement is expressed as a specification statement of the following form. As before, let v be a list of variables, P be a precondition predicate, and Q be a postcondition predicate possibly containing zero-subscripted variables. We use an 'r' subscript to denote parts of requirements.

$$v_r{:}[P_r\ ,\ Q_r]$$

In other words, we are required to achieve property Q_r by modifying variables from list v_r, under the assumption that property P_r holds initially.

A library subroutine consists of both a specification of its capabilities and executable code. Library modules are typically parameterised by variables, types and functions that are specific to the program under construction [9]. Below we usually assume that these (syntactic, macro-like) parameters have already been instantiated, and express the

Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)
1530-0803/05 $20.00 © 2005 **IEEE**

resulting subroutine as a refinement relation of the following form. Let S be an executable statement in our target programming language. We use an 's' subscript to denote parts of library subroutines.

$$v_s{:}[P_s\ ,Q_s] \quad \sqsubseteq \quad S_s$$

This means that code segment S_s can be used to satisfy the requirement expressed by specification $v_s{:}[P_s\ ,Q_s]$. (The refinement relation between the specification and code segment should have been verified when the subroutine was added to the library.)

In general, the goal of formal library 'matching' is to allow requirement $v_r{:}[P_r\ ,Q_r]$ to be replaced by statement S_s, by finding an appropriate relationship between statements $v_r{:}[P_r\ ,Q_r]$ and $v_s{:}[P_s\ ,Q_s]$.

Normally this is allowed under the following circumstances [20]. Let '$v_1 \subseteq v_2$' mean that all variables in list v_1 also appear in list v_2. For two predicates A_1 and A_2, let universal implication $A_1 \Rrightarrow A_2$ mean that predicate $A_1 \Rightarrow A_2$ holds in all states; similarly for equivalence $A_1 \equiv A_2$ and predicate $A_1 \Leftrightarrow A_2$ [14, §2.8]. Let '$E[t/v]$' denote substitution of term(s) t for free occurrences of variable(s) v in expression E [14, §A.2.1].

Adaptation Rule 0 (Perfect match)
Requirement $v_r{:}[P_r\ ,Q_r]$ can be matched with library subroutine $v_s{:}[P_s\ ,Q_s] \sqsubseteq S_s$, and replaced by statement S_s, provided that

0.1) $v_s \subseteq v_r$,

0.2) $P_r \Rrightarrow P_s$, *and*

0.3) $(P_r[v_{r_0}/v_r] \wedge Q_s) \Rrightarrow Q_r$.

In other words, requirement $v_r{:}[P_r\ ,Q_r]$ is satisfied by a library subroutine with capabilities defined by statement $v_s{:}[P_s\ ,Q_s]$ provided that: 0.1) the (instantiated) library subroutine changes no variables other than those that the requirement allows to be updated; 0.2) the library subroutine works correctly in at least as many initial states as the requirement; and 0.3) the library subroutine's behaviour when started in an initial state allowed by the original requirement produces an acceptable behaviour of the original requirement. In proviso 0.3, the (pre-state) variables v_r in predicate P_r are renamed with zero subscripts to correspond with the initial variable naming convention in predicates Q_s and Q_r. In effect, the three provisos on Adaptation Rule 0 guarantee that refinement relationship $v_r{:}[P_r\ ,Q_r] \sqsubseteq v_s{:}[P_s\ ,Q_s]$ holds.

Unfortunately, it is not always possible to find a library subroutine that satisfies all three provisos. Our goal, therefore, is to define rules that adapt subroutine code to allow a match to occur even when one or more of these provisos does not hold. To do this, the rules below add code to the program under construction that compensates for the mismatch between the programmer's requirements and the library subroutine's capabilities. (We assume that a library subroutine whose body 'closely' matches our requirement has already been found. However, quantifying the closeness-of-fit of library components is a separate research topic [10].)

4.1 Avoiding an Inadequate Subroutine

The following adaptation rule is intended to be applied when the chosen library subroutine fails to satisfy Adaptation Rule 0's proviso 0.2. In this case the library subroutine promises to work in only some of the situations anticipated by the requirement. The rule compensates for this shortcoming by introducing a conditional statement to the program so that the subroutine is used only in those situations where it is guaranteed to work [16]. To allow for the remaining situations anticipated by the programmer's requirement, however, the rule also creates a new requirement to be matched in a subsequent step. Let B be a boolean-valued expression in the target programming language. (We also allow B to be used as a predicate.)

Adaptation Rule 1 (Avoidance)
Requirement $v_r{:}[P_r\ ,Q_r]$ can be matched with library subroutine $v_s{:}[P_s\ ,Q_s] \sqsubseteq S_s$, and replaced by statement

if B **then** S_s **else** $v_r{:}[\neg B \wedge P_r\ ,Q_r]$ **end**

provided that

1.1) $v_s \subseteq v_r$,

1.2) $(B \wedge P_r) \Rrightarrow P_s$, *and*

1.3) $((B \wedge P_r)[v_{r_0}/v_r] \wedge Q_s) \Rrightarrow Q_r$.

The provisos are the same as those of Adaptation Rule 0 except that in provisos 1.2 and 1.3 the initial state under which the library code is executed is characterised by predicate '$B \wedge P_r$' instead of just P_r, because we know that B must be true for library code S_s to be reached. In effect, this additional conjunct makes the library subroutine's starting state more specific, and thus makes a successful match easier.

In other anticipated starting states, however, the remaining computational obligation is passed onto a new requirement in the '**else**' part of the statement. This new requirement must achieve the same outcome Q_r as the original one, but this time can do so in the knowledge that condition B is false. (There is, however, no guarantee that the new requirement can itself be satisfied—see the discussion concerning infeasibility in Section 6.)

Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)
1530-0803/05 $20.00 © 2005 **IEEE**

Proof Adaptation Rule 1's proof is trivial. We implicitly rely on the monotonicity of '**if**' statements with respect to refinement [2, p. 56].

$v_r{:}[P_r\ ,\ Q_r]$

$\sqsubseteq$ 'Refinement Law 3, Appendix A'

if B **then** $v_r{:}[B \wedge P_r\ ,\ Q_r]$
else $v_r{:}[\neg B \wedge P_r\ ,\ Q_r]$ **end**

$\sqsubseteq$ 'Adaptation Rule 0 and provisos 1.1 to 1.3'

if B **then** S_s **else** $v_r{:}[\neg B \wedge P_r\ ,\ Q_r]$ **end** $\square$

4.2 Preparing for an Inadequate Subroutine

Another obvious way of compensating for a library subroutine that fails Adaptation Rule 0's proviso 0.2 is to perform a preceding computation that establishes the subroutine's precondition [16]. Let X be a list of logical constants with types consistent with those of the variables in list v_r.

Adaptation Rule 2 (Preparation)
Requirement $v_r{:}[P_r\ ,\ Q_r]$ can be matched with library subroutine $v_s{:}[P_s\ ,\ Q_s] \sqsubseteq S_s$, and replaced by statement

$$v_r{:}[P_r\ ,\ Q_x]\ ;\ S_s$$

provided that

2.1) $v_s \subseteq v_r$,

2.2) $Q_x \Rrightarrow P_s$,

2.3) $(Q_x[X, v_{r_0}/v_{r_0}, v_r] \wedge Q_s) \Rrightarrow Q_r[X/v_{r_0}]$, *and*

2.4) Q_x *contains no zero-subscripted variables other than* v_{r_0}.

New predicate Q_x characterises the program state at the point where the library subroutine begins executing. Proviso 2.2 ensures that this state will satisfy the library subroutine's precondition. Proviso 2.3 then guarantees that executing the library code in this state will achieve the original requirement. Logical constants X are used to capture the initial state of variables v_r in the whole generated program fragment. They do not appear in the generated code, but feature in the provisos as free variables. They are needed because references to 'v_{r_0}' in predicate Q_s will refer to v_r's post-state value in predicate Q_x, rather than v_r's initial value overall. (The constants arise from using Refinement Law 1 in the proof below.) Additional proviso 2.4 is a consequence of the fact that predicate Q_x is used as both the postcondition of the first statement and the precondition of the second, as shown in the proof below.

Proof Adaptation Rule 2's proof proceeds as follows. We implicitly rely on the monotonicity of sequential composition '**;**' with respect to refinement [2, p. 52].

$v_r{:}[P_r\ ,\ Q_r]$

$\sqsubseteq$ 'Refinement Law 1, Appendix A, and proviso 2.4'

con $X \bullet$
$(v_r{:}[P_r\ ,\ Q_x]\ ;\ v_r{:}[Q_x[X/v_{r_0}]\ ,\ Q_r[X/v_{r_0}]])$

$\sqsubseteq$ 'Adaptation Rule 0 and provisos 2.1 to 2.3'

con $X \bullet (v_r{:}[P_r\ ,\ Q_x]\ ;\ S_s)$

$\sqsubseteq$ 'Refinement Law 2, Appendix A'

$v_r{:}[P_r\ ,\ Q_x]\ ;\ S_s$ $\square$

In the second step of the proof, Adaptation Rule 0 expects a proviso of the form '$Q_x[X/v_{r_0}] \Rrightarrow P_s$', but proviso 2.2 omits the substitution. However, recalling that precondition P_s contains no zero-subscripted variables, this renaming of free variables has no effect, so the simpler proviso shown in Adaptation Rule 2 suffices.

4.3 Broadening the Frame

The next rule is intended to be applied when the chosen library subroutine fails to satisfy Adaptation Rule 0's proviso 0.1. In this case the library subroutine wants the freedom to change some variables v_ℓ that the requirement does not allow to be updated.

The following rule solves this by declaring variables v_ℓ locally, so that they can be modified by the subroutine without affecting the surrounding program. These newly-declared local variables are initialised in such a way that they make the precondition of the library subroutine true. Let variable(s) v_ℓ and expression(s) E be of type(s) T. Let E_0 abbreviate substitution $E[v_{r_0}/v_r]$ [14, p. 113]. Let 'v_r, v_ℓ' be the list comprising sublists v_r and v_ℓ.

Adaptation Rule 3 (Broadening)
Requirement $v_r{:}[P_r\ ,\ Q_r]$ can be matched with library subroutine $v_s{:}[P_s\ ,\ Q_s] \sqsubseteq S_s$, and replaced by statement

$$\textbf{var}\ v_\ell : T \bullet (v_\ell\ \textbf{:=}\ E\ ;\ S_s)$$

provided that

3.1) $v_s \subseteq v_r, v_\ell$,

3.2) $P_r \Rrightarrow P_s[E/v_\ell]$,

3.3) $(P_r[v_{r_0}/v_r] \wedge Q_s[E_0/v_{\ell_0}]) \Rrightarrow Q_r$,

3.4) *variable lists v_ℓ and v_r are disjoint, and*

3.5) *variables v_ℓ and v_{ℓ_0} do not appear free in expressions E, P_r or Q_r.*

Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)
1530-0803/05 $20.00 © 2005 **IEEE**

In this case provisos 3.2 and 3.3 incorporate the knowledge that fresh variables v_ℓ have been initialised to E. Proviso 3.5 tells us that these new variables should not appear in the original requirement. It also says that they may not appear in expression E; this is reasonable because any occurrences of 'v_ℓ' in E would refer to uninitialised values.

Proof Adaptation Rule 3's proof proceeds as follows. The key step is the second one, which sets up introduction of the leading assignment statement by modelling its effect with an equivalence. In particular, the second proof step takes advantage of the fact that the v_ℓ variables are fresh to eliminate the substitutions required by Refinement Rule 1.

$v_r{:}[P_r\ ,\ Q_r]$

$\sqsubseteq$ 'Refinement Law 4, Appendix A, and proviso 3.4'

var $v_\ell : T \bullet v_r, v_\ell{:}[P_r\ ,\ Q_r]$

$\sqsubseteq$ 'Refinement Law 1, Appendix A, and proviso 3.5'

var $v_\ell : T \bullet$ (**con** $X \bullet (v_\ell{:}[P_r\ ,\ P_r \wedge v_\ell = E]$ **;** $v_r, v_\ell{:}[P_r \wedge v_\ell = E\ ,\ Q_r]$))

$\sqsubseteq$ 'Refinement Law 2, Appendix A'

var $v_\ell : T \bullet (v_\ell{:}[P_r\ ,\ P_r \wedge v_\ell = E]$ **;** $v_r, v_\ell{:}[P_r \wedge v_\ell = E\ ,\ Q_r])$

$\sqsubseteq$ 'Refinement Law 5, Appendix A'

var $v_\ell : T \bullet (v_\ell$ **:=** E **;** $v_r, v_\ell{:}[P_r \wedge v_\ell = E\ ,\ Q_r])$

$\sqsubseteq$ 'Adaptation Rule 0 and provisos 3.1 to 3.3'

var $v_\ell : T \bullet (v_\ell$ **:=** E **;** $S_s)$ $\quad\square$

In the final proof step, Adaptation Rule 0 expects a proviso of the form '$(P_r \wedge v_\ell = E) \Rrightarrow P_s$', where we have used proviso 3.2. However, keeping in mind that variables v_ℓ are not mentioned in precondition P_r, these two forms of proviso are logically equivalent. (We prefer to use substitutions in the rules' provisos because syntactic renamings are slightly easier to manipulate during theorem proving than adding an equivalence as a hypothesis.) Similarly, the final proof step expects '$(P_r[v_{r_0}/v_r] \wedge v_{\ell_0} = E[v_{r_0}/v_r] \wedge Q_s) \Rrightarrow Q_r$', where we have used simpler proviso 3.3. Again, the two provisos are equivalent due to the fact that variables v_{ℓ_0} do not appear free in predicates P_r and Q_r.

4.4 Completing an Inadequate Subroutine

Our final adaptation rule is intended for use when Adaptation Rule 0's proviso 0.3 fails. It compensates by following the library subroutine's code with a statement that completes the unfinished requirement. It also incorporates Adaptation Rule 2 by preceding the library code with another new requirement in order to guarantee that the library subroutine will terminate from all states anticipated by the original requirement. (The new preceding requirement could be equivalent to the null statement, **skip** [14, p. 12], so the rule below can be used to add code only following the library subroutine, if desired.) Let P_x and Q_x be predicates which may contain zero-subscripted variables—despite their nomenclature, they both occur as pre and postconditions. Their role is to act as the pre and postcondition, respectively, to be matched against the library subroutine.

Adaptation Rule 4 (Completion)
Requirement $v_r{:}[P_r\ ,\ Q_r]$ can be matched with library subroutine $v_s{:}[P_s\ ,\ Q_s] \sqsubseteq S_s$, and replaced by statement

con $X \bullet (v_r{:}[P_r\ ,\ P_x]$ **;** S_s **;** $v_r{:}[Q_x[X/v_{r_0}]\ ,\ Q_r[X/v_{r_0}]])$

provided that

4.1) $v_s \subseteq v_r$,

4.2) $P_x \Rrightarrow P_s$,

4.3) $(P_x[X, v_{r_0}/v_{r_0}, v_r] \wedge Q_s) \Rrightarrow Q_x[X/v_{r_0}]$, *and*

4.4) P_x *and* Q_x *contain no zero-subscripted variables other than* v_{r_0}.

Logical constants X in the generated code serve to capture the initial values of variables v_r so that they can be accessed by the third sequentially-composed statement. Fortunately, as demonstrated by the example in Section 5.1, these constants can be eliminated as soon as the specification statement that refers to them is replaced with executable code.

Proof Adaptation Rule 4's proof is made simple by taking advantage of our previous proof of Adaptation Rule 2.

$v_r{:}[P_r\ ,\ Q_r]$

$\sqsubseteq$ 'Refinement Law 1, Appendix A, and proviso 4.4'

con $X \bullet$
$(v_r{:}[P_r\ ,\ Q_x]$ **;** $v_r{:}[Q_x[X/v_{r_0}]\ ,\ Q_r[X/v_{r_0}]])$

$\sqsubseteq$ 'Adaptation Rule 2 and provisos 4.1 to 4.3'

con $X \bullet$
$(v_r{:}[P_r\ ,\ P_x]$ **;** S_s **;**
$v_r{:}[Q_x[X/v_{r_0}]\ ,\ Q_r[X/v_{r_0}]])$ $\quad\square$

A subtle point about this proof is that the logical constants X in proviso 4.3, which derived from application of Adaptation Rule 2, are not the same constants X as those remaining in the generated code, which derived from application of Refinement Law 1. Nevertheless, both lists of constants serve to characterise the same initial state. That we therefore chose to use the same name 'X' for both does not create any conflicts; those in the generated code are bound by the '**con**' declaration, whereas those in the proviso are free identifiers used to achieve renaming.

Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)
1530-0803/05 $20.00 © 2005 **IEEE**

5 Examples

To demonstrate the adaptation theory, this section presents two simple examples which, together, use all of the rules above (Adaptation Rule 2 is subsumed by Rule 4).

In practice, however, we would not normally expect the semantic theory above to be applied directly in this way, since the provisos on the rules introduce the need for theorem proving. Instead, the semantic rules above are intended to provide a basis for proving the correctness of special-case 'syntactic' rules, which can be applied mechanically, with as little creative input from the programmer as possible (in the same way that refinement theory introduces easily-applied syntactic rules, justified by refinement's semantic foundation [14, Ch. 21]).

5.1 Calculating a Remainder

Consider the following requirement. Let the '**rem**' operator return the remainder after integer division, and assume that x and y have been declared as integers.

$$y{:}[x > 0 \,,\, y = y_0 \,\mathbf{rem}\, x]$$

Thus, given a positive divisor x, the aim is to set y equal to the remainder of dividing itself by x. We further assume that the available library module for integer arithmetic does not contain a matching subroutine, but that the closest match to this requirement is as follows (with the library subroutine's variable-valued parameters [9] already instantiated with 'x' and 'y').

(1) $\quad y{:}[x > 0 \wedge y \geqslant 0 \,,\, y = y_0 \,\mathbf{rem}\, x]$
$\quad\sqsubseteq\quad$ **while** $y > x$ **do** y **:=** $y - x$ **end**

Library subroutine 1 calculates the remainder by repeated subtraction. Unfortunately, its precondition reveals that it will not work correctly if dividend y is negative—in which case the loop never terminates—whereas the programmer's requirement does not guarantee this.

Nevertheless, we can use our adaptation rules to overcome the library subroutine's shortcomings.

$y{:}[x > 0 \,,\, y = y_0 \,\mathbf{rem}\, x]$
$\sqsubseteq$ 'Adaptation Rule 1 and library subroutine 1'
if $y \geqslant 0$ **then**
  while $y > x$ **do** y **:=** $y - x$ **end**
else
  $y{:}[y < 0 \wedge x > 0 \,,\, y = y_0 \,\mathbf{rem}\, x]$
end

Thus the library subroutine is used in the conditional statement's '**then**' part, protected by the condition that the dividend y is non-negative in this case. Provisos 1.1 to 1.3 all hold straightforwardly in this step.

However, we must still satisfy the new requirement introduced in the '**else**' alternative. To do this we make use of the general rule for completing computations, and the same library subroutine.

$y{:}[y < 0 \wedge x > 0 \,,\, y = y_0 \,\mathbf{rem}\, x]$
$\sqsubseteq$ 'Adaptation Rule 4 and library subroutine 1'
con X •
  ($y{:}[y < 0 \wedge x > 0 \,,\, y = -y_0 \wedge y > 0 \wedge x > 0]$ **;**
  while $y > x$ **do** y **:=** $y - x$ **end ;**
  $y{:}[y = -(X \,\mathbf{rem}\, x) \,,\, y = (X \,\mathbf{rem}\, x)]$)

The creative part of this step is to identify the new pre and postconditions surrounding the library code. With respect to Adaptation Rule 4, the new precondition P_x here is '$y = -y_0 \wedge y > 0 \wedge x > 0$' and the new postcondition Q_x is '$y = -(y_0 \,\mathbf{rem}\, x)$'. The first of these choices was informed by the observation that $y < 0 \Rightarrow -y > 0$, which suggests how to satisfy the library subroutine's precondition. The choice of postcondition was then made by observing that $(-y) \,\mathbf{rem}\, x = -(y \,\mathbf{rem}\, x)$ which tells us what state the library subroutine will achieve.

Adaptation Rule 4's provisos can all be proven easily here. Proviso 4.2 is immediate.

$$y = -y_0 \wedge y > 0 \wedge x > 0 \quRightarrow\quad x > 0 \wedge y \geqslant 0$$

Proviso 4.3 is proven as follows.

$(y = -y_0 \wedge y > 0 \wedge x > 0)[X, y_0/y_0, y] \wedge$
$y = y_0 \,\mathbf{rem}\, x$
$\equiv\quad y_0 = -X \wedge y_0 > 0 \wedge x > 0 \wedge y = y_0 \,\mathbf{rem}\, x$
$\Rrightarrow\quad y = (-X) \,\mathbf{rem}\, x$
$\equiv\quad y = -(X \,\mathbf{rem}\, x)$
$\equiv\quad (y = -(y_0 \,\mathbf{rem}\, x))[X/y_0]$

This leaves two simple requirements to be matched. Assume the integer arithmetic library module contains the following (instantiated) subroutine for negating an integer.

(2) $\quad y{:}[\text{true} \,,\, y = -y_0] \quad\sqsubseteq\quad y$ **:=** $-\,y$

This can be used directly on the first requirement. Provisos 0.1 to 0.3 are all straightforward here.

$y{:}[y < 0 \wedge x > 0 \,,\, y = -y_0 \wedge y > 0 \wedge x > 0]$
$\sqsubseteq$ 'Adaptation Rule 0 and library subroutine 2'
y **:=** $-\,y$

Similarly for the second requirement, to yield the same assignment statement. At this point, there are no remaining references to constants X, so their declaration can be removed using Refinement Law 2 from Appendix A.

Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)
1530-0803/05 $20.00 © 2005 **IEEE**

Thus, the final executable program, generated from the original requirement via a library module containing imperfectly-matching subroutines, is as follows.

if $y \geqslant 0$ **then**
  while $y > x$ **do** y **:=** $y - x$ **end**
else
  y **:=** $-\,y$ **;**
  while $y > x$ **do** y **:=** $y - x$ **end ;**
  y **:=** $-\,y$
end

5.2 Finding the Largest Array Element

Consider the following program under construction. It is intended to set variable m equal to the maximum value in an array a of natural numbers (or zero if the array is empty). Variable i is used as the loop counter. Let value L denote the length of the array.

i **:=** 1 **;**
m **:=** 0 **;**
while $i \leqslant L$ **do**
  m:[true , $m = \max(a(i), m_0)$] **;**
  i **:=** $i + 1$
end

For the purposes of illustration, we assume that the target programming language does not have an in-built '$\max$' operator.

Therefore, to satisfy the requirement inside the loop, we must search the software library for the closest match to the above requirement. In this case the following subroutine for ordering two numbers is found. The subroutine template [9] in the library is parameterised by the names of three program variables, α, β and γ, in that order, where 'γ' must be a previously-undeclared name. Let $\mathbb{N}$ be the natural number type.

(3) α, β:[true , $\alpha = \min(\alpha_0, \beta_0) \wedge \beta = \max(\alpha_0, \beta_0)$]
$\sqsubseteq$ **if** $\beta < \alpha$ **then**
  var $\gamma : \mathbb{N} \bullet$ (γ **:=** α **;**
    α **:=** β **;**
    β **:=** γ)
  end

Thus the subroutine template ensures that α is less than or equal to β, and uses temporary variable γ to swap their values if this is not the case.

Although this template does not match our requirement, we can nevertheless instantiate it and use Adaptation Rule 3 to make it suitable. Assume that variable names 'ℓ' and 't' have not been used previously.

m:[true , $m = \max(a(i), m_0)$]
$\sqsubseteq$ 'Adaptation Rule 3 and subroutine template 3 with 'α' mapped to 'ℓ', 'β' mapped to 'm' and 'γ' mapped to 't' '
var $\ell : \mathbb{N} \bullet$ (ℓ **:=** $a(i)$ **;**
  if $m < \ell$ **then**
    var $t : \mathbb{N} \bullet$ (t **:=** ℓ **;**
      ℓ **:=** m **;**
      m **:=** t)
  end)

Provisos 3.1, 3.2, 3.4 and 3.5 are all trivial in this case. Proviso 3.3 is proven as follows.

$\text{true}[m_0/m] \wedge$
$(\ell = \min(\ell_0, m_0) \wedge$
$\ m = \max(\ell_0, m_0))[(a(i))[m_0/m]/\ell_0]$
$\equiv$ $\ell = \min(a(i), m_0) \wedge m = \max(a(i), m_0)$
$\Rrightarrow$ $m = \max(a(i), m_0)$

Putting this newly-instantiated subroutine back into the original context yields the following program for finding the largest element in an array.

i **:=** 1 **;**
m **:=** 0 **;**
while $i \leqslant L$ **do**
  var $\ell : \mathbb{N} \bullet$ (ℓ **:=** $a(i)$ **;**
    if $m < \ell$ **then**
      var $t : \mathbb{N} \bullet$ (t **:=** ℓ **;**
        ℓ **:=** m **;**
        m **:=** t)
    end) **;**
  i **:=** $i + 1$
end

This program is far from optimal. However, it represents a better outcome than not being able to generate any code at all from the non-matching library module.

6 Discussion

Adaptation Rules 1, 2 and 4 all introduce new requirements which must be satisfied in subsequent steps. However, it is possible that the new requirements so created are *infeasible* [14, §1.7], which means that they cannot be satisfied by any program. In practice, infeasible requirements are development 'dead ends' from which the programmer must ultimately backtrack.

Notably absent from our collection of rules is one which generates iterative code. (However, the example in Section 5.1 showed how iteration may occur within a library

Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)
1530-0803/05 $20.00 © 2005 **IEEE**

subroutine, and the example in Section 5.2 showed how library code may occur within an iterative construct.) Indeed, an adaptation rule which uses library subroutine '$v_s{:}[P_s\ ,\ Q_s] \sqsubseteq S_s$' to replace requirement $v_r{:}[P_r\ ,\ Q_r]$ with code of the form '**while** B **do** S_s **end**' would sit awkwardly in our set of rules. In this situation the requirement refers to the overall effect of the loop (e.g., to increment all elements of an array), whereas the library subroutine refers to the effect of one iteration only (e.g., to increment a single number). Thus there is a fundamental 'mismatch' between the requirement and subroutine. Previous work in the Component-Based Software Engineering literature has recognised this and typically treats 'iterator' and 'accumulator' constructors specially [8]. (The rules above are still 'complete' in the sense that they compensate for the failure of all three provisos associated with basic library subroutine matching [20].)

7 Conclusion

We have developed a set of formal rules for adaptation of software library subroutine code, based on program refinement theory. This outcome provides a sound basis for correctness proofs of 'adaptation' code introduced during the construction of safety or mission-critical software.

There are two ways in which this work can be extended. Firstly, as noted above, our semantic rules can be used to verify special-case rules which can be applied syntactically, in just the same way that syntactic refinement rules are derived from semantic refinement principles [14, Ch. 21]. The resulting 'syntactic adaptation rules' could then be applied automatically, using pattern matching and unification, with relatively little creative input from the programmer.

Secondly, while this paper defines a theory for adapting individual subroutines from a library module, we can also envisage adaptation of library modules as a whole, simultaneously changing both their encapsulated data structures and subroutines in a consistent way. Indeed, such 'module adaptation rules' would build directly on the theory above, in the same way that simple refinement theory is extended to 'data refinement' [14, §21.3.10].

Acknowledgements We wish to thank Bertrand Meyer for his helpful comments on this paper, David Hemer for his advice on software library templates, and the anonymous reviewers of drafts of this paper. This research was funded by Australian Research Council Discovery Grant DP0208046, *Compilation of Specifications*.

References

[1] R. B. Ayed, J. Desharnais, M. Frappier, and A. Mili. A calculus of program adaptation and its applications. *Science of Computer Programming*, 38(1–3):73–123, August 2000.

[2] R.-J. R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (REX Workshop 1989)*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer-Verlag, 1989.

[3] A. Bracciali, A. Brogi, and C. Canal. Systematic component adaptation. In A. Brogi and E. Pimentel, editors, *Proceedings of the Workshop on Formal Methods and Component Interaction*, volume 66 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[4] M. Büchi and E. Sekerinski. Formal methods for component software: The refinement calculus perspective. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proceedings of the Second Workshop on Component-Oriented Programming (WCOP)*, pages 23–32. TUCS General Publication 5, 1997.

[5] M. Frappier, A. Mili, and J. Desharnais. A relational calculus for program construction by parts. *Science of Computer Programming*, 26:237–254, 1996.

[6] A. Frick, W. Zimmer, and W. Zimmermann. On the design of reliable libraries. In *Technology of Object-Oriented Programming (TOOLS-17)*, pages 13–23. Prentice-Hall, 1995.

[7] C. Haack, B. Howard, A. Stoughton, and J. Wells. Fully automatic adaptation of software components based on semantic specifications. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology (AMAST 2002)*, volume 2422 of *Lecture Notes in Computer Science*, pages 83–98. Springer-Verlag, 2002.

[8] M. Heaney. Charles: A data structure library for Ada95. In J.-P. Rosen and A. Strohmeier, editors, *Reliable Software Technologies—Ada-Europe 2003*, volume 2655 of *Lecture Notes in Computer Science*, pages 271–282. Springer-Verlag, 2002.

[9] D. Hemer. Higher-order associative commutative pattern matching for component retrieval. In M. Atkinson, editor, *Computing: The Australasian Theory Symposium (CATS'04)*, volume 91 of *Electronic Notes in Theoretical Computer Science*, pages 175–191. Elsevier, 2004.

[10] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank:

Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)
1530-0803/05 $20.00 © 2005 **IEEE**

Relative significance rank for software component search. In L. Clarke, L. Dillon, and W. Tichy, editors, *25th International Conference on Software Engineering (ICSE 2003)*, pages 14–24. IEEE Computer Society Press, 2003.

[11] P. Inverardi and M. Tivoli. Software architecture for correct components assembly. In M. Bernardo and P. Inverardi, editors, *Formal Methods for Software Architectures (SFM 2003)*, volume 2804 of *Lecture Notes in Computer Science*, pages 92–121. Springer-Verlag, 2003.

[12] Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, August 1992.

[13] R. McKegney and T. Shepard. Techniques for embedding executable specifications in software component interfaces. In H. Erdogmus and T. Weng, editors, *International Conference on COTS-Based Software Systems (ICCBSS'03)*, volume 2580 of *Lecture Notes in Computer Science*, pages 143–156. Springer-Verlag, 2003.

[14] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[15] C. O'Halloran. Issues for the automatic generation of safety critical software. In *The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, pages 277–280. IEEE Computer Society Press, 2000.

[16] J. Penix and P. Alexander. Toward automated component adaptation. In N. Juristo, editor, *Proceedings of the Ninth International Conference on Software Engineering and Knowledge Engineering (SEKE'97)*, pages 535–542. Knowledge Systems Institute, June 1997.

[17] P. S. Roop, A. Sowmya, and S. Ramesh. Forced simulation: A technique for automating component reuse in embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 6(4):602–628, October 2001.

[18] D. R. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990. Special Issue on Formal Methods.

[19] B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In L. Clarke, L. Dillon, and W. Tichy, editors, *25th International Conference on Software Engineering (ICSE 2003)*, pages 374–384. IEEE Computer Society Press, 2003.

[20] A. Zaremski and J. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.

A Refinement Laws Used in Proofs

The following refinement laws were used in the proofs in Section 4. Let S be a statement in our modelling language; B be a boolean-valued target-language expression; P be a predicate on the system state; Q be a predicate on the system state which may contain variables decorated with zero subscripts; v be a list of variable identifiers; and T be a list of types.

Refinement Law 1 (Sequential composition) [14, Law 6.11]

$$
\begin{array}{ll}
 & v_1, v_2{:}[P\ ,\ Q_1] \\
\sqsubseteq & \text{`for previously-unused constants } X\text{, and provided that } Q_2 \text{ contains no zero-subscripted variables other than } v_{1_0}\text{'} \\
 & \textbf{con } X \bullet \\
 & \quad (v_1{:}[P\ ,\ Q_2]\ \textbf{;}\ v_1, v_2{:}[Q_2[X/v_{1_0}]\ ,\ Q_1[X/v_{1_0}]])
\end{array}
$$

Refinement Law 2 (Remove constant) [14, Law 6.4]

$$
\begin{array}{ll}
 & \textbf{con } X \bullet S \\
\sqsubseteq & \text{`provided } X \text{ does not appear in statement } S\text{'} \\
 & S
\end{array}
$$

Refinement Law 3 (Choice) [14, Special case of Law 5.1]

$$
\begin{array}{ll}
 & v{:}[P\ ,\ Q] \\
\sqsubseteq & \textbf{if } B \textbf{ then } v{:}[B \wedge P\ ,\ Q] \textbf{ else } v{:}[\neg B \wedge P\ ,\ Q] \textbf{ end}
\end{array}
$$

Refinement Law 4 (Variable declaration) [14, Special case of Law 3.2]

$$
\begin{array}{ll}
 & v_1{:}[P\ ,\ Q] \\
\sqsubseteq & \text{`provided lists } v_1 \text{ and } v_2 \text{ are disjoint'} \\
 & \textbf{var } v_2 : T \bullet v_1, v_2{:}[P\ ,\ Q]
\end{array}
$$

Refinement Law 5 (Assignment) [14, Law 6.6]

$$
\begin{array}{ll}
 & v_1, v_2{:}[P\ ,\ Q] \\
\sqsubseteq & \text{`provided } (v_1 = v_{1_0} \wedge P) \Rrightarrow Q[E/v_1]\text{'} \\
 & v_1 := E
\end{array}
$$

Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)
1530-0803/05 $20.00 © 2005 **IEEE**